CONFORMANCE EXECUTION OF NON-DETERMINISTIC SPECIFICATIONS FOR COMPONENTS

TECHNICAL FIELD

The invention relates to testing program code, and more particularly to verifying the execution of program code in conformance with a behavioral specification.

BACKGROUND

One of the most important and challenging aspects of software development is testing. Testing involves determining whether software executes in a manner consistent with the software's intended behavior. The software's intended behavior may be defined using an executable specification language such as the Abstract State Machine Language (AsmL). AsmL may be employed to specify precise conforming behavior (a deterministic specification), or to specify ranges or choices within which various acceptable behaviors may take place (a non-deterministic specification). Non-deterministic specifications are desirable because they do not specify the implementation behavior down to the finest detail. Non-deterministic specifications define choices for behavior, allowing the software implementer design freedom within those choices. An AsmL specification may be executed by itself (in a stand-alone manner) or in conjunction with a separate implementation in order to ascertain whether or not the behavior defined by the specification is in fact the desired behavior.

A software implementation's runtime behavior may be compared against the specified behavior to identify non-conformities. This process is referred to as conformance testing. One approach to conformance testing is to insert code into the software to test

conditions. The conditions define the desired state of the software at method boundaries of the software. For examples of this approach, see Bertrand Meyer, Eiffel: The Language, Object Oriented Series, Prentice Hall, New York, NY, 1992. See also the White Paper by Murat Karaorman et al. of Texas Instruments et al., entitled jContractor: A Reflective Java Library to Support Design By Contract, and the White Paper by Reto Kramer of Cambridge Technology Partners, entitled iContract – The Java Design by Contract Tool.

A limitation of existing methods of conformance testing is that they do not provide adequate support for non-deterministic specifications. Another limitation of existing methods is that they do not adequately address the conformance checking of call sequences. A software specification may indicate that processing within a method of an implementing class should proceed in steps, e.g. in a particular order. The specification may also indicate that the implementation method must make calls to methods of another class (so called 'mandatory calls'). The mandatory calls may lead to re-entrance of methods of the implementing class, resulting in unpredictable state changes while the mandatory calls are still pending. Existing approaches to conformance checking do not adequately address this situation.

SUMMARY

To perform conformance checking of a software implementation with a nondeterministic specification, a software implementation and a software specification are applied to produce a conformance-test (CT) enabled implementation. Nondeterministic choices of the software specification result in assigning a corresponding choice of the CT enabled implementation to a variable. The CT enabled implementation includes a test that

the variable then comprises one of the nondeterministic choices allowed by the software specification.

To perform conformance testing where the software specification includes ordered steps, and calls to methods of other classes (mandatory calls), a software object is produced and organized such that each step of the software specification has a corresponding code section in the software object. The software object includes instructions to generate an identification of a mandatory call comprised by the software specification, and instructions to test that the state of the implementation conforms to the software specification during the mandatory call.

DRAWINGS

Figure 1 is a block diagram of a system embodiment of an execution environment.

Figure 2 is a block diagram of a system embodiment to perform software conformance testing.

Figure 3 is an embodiment of an AsmL code listing of an implementation-specific declarative software specification.

Figure 4 is an embodiment of a C# code listing of a conformance-test enabled implementation for the specification of Figure 3.

Figure 5 is an embodiment of an AsmL code listing of a declarative interface specification.

Figure 6 is an embodiment of a C# code listing to synchronize variables of an implementation and the specification of Figure 5.

Figure 7 is an embodiment of an AsmL code listing of an operational software specification.

Figures 8 and 9 are embodiments of C# code listings of a conformance-test enabled implementation for conformance testing with the specification of Figure 7.

Figure 10 is an embodiment of a C# code listing to synchronize variables of an implementation and the specification of Figure 7.

Figure 11 is an embodiment of an AsmL code listing of a nondeterministic operational software specification.

Figure 12 is an embodiment of a C# code listing of a conformance-test enabled implementation for conformance testing with steps and mandatory calls of the software specification of Figure 7.

Figure 13 is an embodiment of a C# code listing of a class for conformance testing with steps and mandatory calls of the software specification of Figure 7.

Figures 14-16 are an embodiment of a C# code listing of a method of the class of Figure 13 for conformance testing with steps and mandatory calls of the software specification of Figure 7.

Figure 17 is an embodiment of a C# code listing of the target of a mandatory call method to assist conformance checking with the specification of Figure 7.

Figure 18 is a block diagram of an apparatus embodiment to perform conformance checking.

DESCRIPTION

In the following figures and description, like numbers refer to like elements. References to "one embodiment" or "an embodiment" do not necessarily refer to the same embodiment, although they may.

A software implementation in the form of a class definition may be compiled into an intermediate language (IL) form. Likewise, a specification corresponding to the software implementation may be compiled into an IL form. The IL forms of the implementation and the specification may be applied to produce a conformance-tested IL form of the implementation that may be executed in a run-time environment. Simply executing the conformance-tested IL produces an error, assertion, or other indication when the implementation does not conform to the specification.

With reference to the system embodiment 100 of Figure 1, software source code 102 is compiled into an "intermediate language" (IL) form 104 before being loaded into a runtime environment 106. The IL form 104 may be referred to as the implementation IL. The source code 102 is typically a high-level programming language form such as C++, C# (pronounced C sharp), Visual Basic, and so on. The implementation IL 104 includes platform-neutral instructions that may be executed by a virtual machine (VM) of the runtime environment 106. Platform neutral instructions are instructions that are not specific to a particular hardware processor or hardware configuration. One example of an intermediate language is the Microsoft Intermediate Language (MSIL). The VM of the runtime environment 106 abstracts the underlying hardware platform to provide a platform-neutral environment for executing the implementation IL 104. In addition to the VM, the runtime

environment 106 may include I/O facilities and garbage collection, among other things. The Microsoft .NET Runtime Environment is an example of a runtime environment 106 that supports the execution of MSIL.

The software specification and the software implementation can be in any languages. The languages of the specification and implementation need not be the same. In one embodiment, the specification is in the Abstract State Machine Language (AsmL), although this need not be the case. Once the specification and implementation are compiled to a common IL, portions of code from both may be applied to produce the conformance-test enabled implementation code. Producing and executing a single body of code alleviates difficulties that arise from separately executing the implementation IL and the specification IL and then attempting to compare the results of the separate executions.

With reference to the system embodiment 200 of Figure 2, an AsmL software specification 202 is compiled to an intermediate language form 204, which may be referred to as the specification IL. The specification IL 204 is applied to the implementation IL 104 to produce a conformance test enabled intermediate language form 206 (CT enabled IL form). The CT enabled IL form 206 may be loaded and executed by the runtime environment 106 to conformance test the behavior of the implementation IL 104 with the behavior defined by the specification 202.

Broadly, two types of specifications are possible. One type of specification is declarative. A declarative specification specifies the behavior of an implementation in terms of logical conditions that must prevail at different points during the execution of the implementation. Another type of specification is operational. An operational specification

defines the behavior of an implementation in terms of actions that the implementation should take. Both types of specifications can be specific to a particular implementation class, or more general so that they apply to any class that implements one or more specified methods. The latter is referred to herein as an interface specification. Interface specifications may be reused with different implementation classes, but they tend to be more abstract and hence somewhat more complex than implementation-specific specifications. Figure 3 illustrates an embodiment of a declarative specification that is specific to a particular implementation class, and Figure 4 illustrates an embodiment of a corresponding CT enabled implementation class. Figure 5 illustrates an embodiment of a declarative interface specification, i.e., a declarative specification that is not specific to a particular implementation class. Figure 6 illustrates additional (beyond what is found in Figure 4) "abstraction" code required by a CT enabled implementation class for the declarative interface specification of Figure 5.

Figure 3 is a code listing embodiment for the specification of a hash table. A hash table is data and associated methods to associate unique keys with values. One way to implement a hash table is using two arrays; the first array holds the keys, and the second array holds the values associated with the keys. Figure 3 is a code listing of an embodiment 300 of a class to specify the behavior of a hash table implementation.

An AsmL specification for a hash table begins with the class declaration 302. The declaration 302 inherits from the implementation class (Hashtable). In other words, the specification inherits from the implementation for which it is a specification. This situation provides the specification with access to all of the protected state (member) variables of the

implementation. Public member variables are accessible from any class, and private member variables are not accessible from any other class. Protected member variables are accessible to classes derived from the class, but not to unrelated classes.

Lines 304-314 specify a 'constraint'. A constraint is a state condition that must be true for the implementation class at all times in order to avoid an error situation. In this case, the constraint indicates that at all times:

- The arrays that hold the keys and values of the Hashtable implementation class should have the same length and should not be null.
- No key should have a value that is null.
- No two keys in the key array should be identical.

Lines 319 to 326 specify an 'ensure' condition for the set() method of the Hashtable class. The set() method sets (associates) a key-value pair in the hash table. To comply with the specification, the ensure condition must hold true at the conclusion of the implementation's set() method. In this case, the ensure condition specifies:

- If the key provided to the method call is null, the method throws an ArgumentNullException.
- The keys and values in corresponding positions of the arrays are set to the key and value provided in the method call.
- The return value of the method should be the value provided in the method call.

Conditions that must hold upon entry to a method are specified with a 'require' clause. The specification 300 does not comprise any require clauses. The specification 300 is

an example of a declarative specification. The desired behavior of the implementation is specified in terms of the conditions that must be met at all times (constraint clauses), conditions that must be met when a method is entered (require clauses), and conditions that must be met when a method is about to return (ensure clauses).

With reference to Figure 4, the specification code 300 and the implementation code are applied to produce conformance-test enabled code 400 (CT enabled code). The CT enabled code is a body of code that executes the implementation and in the process tests the implementation for conformance with the specification. When the implementation does not conform to the specification, an error arises. The error may be provided by well-known techniques such as assertions, indicated in the figures by ASSERT. Other techniques for reporting errors, such as log files, could also be used. For simplicity, the CT enable code 400 is illustrated as high-level source and pseudo-code. In one embodiment, the CT enabled code 400 is produced in MSIL. Language within double brackets [[]] identifies code to be inserted. Language within angle brackets <> identifies a substitution of new code for existing code. The substitution is identified as follows: old / new, where 'old' is the code to replace, and new is the code that replaces it.

The class declaration 402 implements an interface called IDictionary. The IDictionary interface may specify general methods for acting upon a 'dictionary', i.e., a collection of key-value pairs. A hash table is a specific implementation of a dictionary.

Line 406 defines the Hashtable$Invariant() method that enforces the constraint condition of the specification code 300. Whenever the method Hashtable$Invariant() is

called, it checks that the constraint condition is satisfied and cause an error (by way of an assertion) otherwise.

Line 407 defines the set$Pre() method that implements any specified require clause. No require clause was specified, and so the Set$Pre() method, when called, simply asserts true (i.e., it never causes an error).

Lines 408-412 define the set$Post() method that enforces the specified ensure clause. The set$Post() checks (by way of an assertion) that the ensure condition is satisfied and causes an error otherwise.

Line 414 declares the key and value arrays for the hash table implementation. Lines 418-450 implement the set() method of the implementation, with conformance checking. A return object is declared at 420, and at line 422, the Hashtable$Invariant() method is called to check the constraint condition (which must always hold within the method). At 424 the set$Pre() method is called. If a require condition is specified, the require condition is enforced by this call.

Lines 426-438 define a try-catch construct for exception handling. The body of the try clause (lines 428-432) is executed, and if an exception results, control is transferred to the catch clause (lines 434-438). When an exception takes place, line 436 assigns the exception to a variable. The body of the set() method from the implementation is inserted at 428. Line 432 indicates that all return statements in the body of the set method from the implementation are replaced with the following code:

result = value; break END

In other words, instead of returning the value that was set, the method now assigns the value to a variable and jumps to line 440, which is labeled END. At line 440, the variable result either contains the return value assigned in the try clause, or the exception assigned in the catch clause. At 444 the Hashtable$Invariant() method is called to enforce the constraint condition before the method returns. At 446 the set$Post() method is called to enforce the ensure condition. If the ensure condition is not satisfied, set$Post() causes an error.

At line 448, the result variable is checked to determine whether it holds an exception or a return value. If result holds an exception, the exception is thrown. Otherwise, the result is returned. (In this embodiment, it is assumed that no exception type is ever returned as a normal result.)

One limitation of the specification 300 of Figure 3 is that it is specific to the Hashtable implementation class, and in fact extends the implementation class. A more 'abstract' specification that defines the behavior of implementations independently of a particular class definition could be applied to any implementing class. One manner of specifying behavior independently of the implementation is to avoid the use of specific implementation variables in the specification. (The specification 300 of Figure 3 was specific to the Hashtable class because it inherited from and employed class variables of Hashtable). With reference to the code listing embodiment 500 of Figure 5, a specification that is independent of a particular implementation class begins at 502 with a class declaration. The class IDictionary$Contract specifies the IDictionary interface. At 504 a map object is declared. The map object associates values or objects with corresponding values or objects in a one-to-one fashion. Lines 512 to 514 define an ensure condition for the

set method of any class that implements the IDictionary interface. Unlike the specification 300 of Figure 3, the ensure condition of Figure 5 is not defined in terms of particular implementation variables. Instead, the ensure condition is specified in terms of the specification map variable:

- If the key provided to the method is null, the method throws an ArgumentNullException.
- The updated map should associate the provided key with the provided value, and the method should return the provided value.

The conditions of the specification are defined in terms of the map variable. However, implementations do not use this variable, but instead use variables typically chosen to optimize some implementation constraint, such as processing speed or memory size. In order to synchronize the use of specification variables in the specification, and implementation variables in the implementation, an implementation constructs an instance of the specification class IDictionary$Contract. The specification class is constructed using implementation variables. With reference to the code listing embodiment 600 of Figure 6, at 604 to 608 an implementation class may define a method called abstraction() to construct, initialize, and return an instance of the IDictionary$Contract class. The IDictionary$Contract instance comprises specification variables initialized from the implementation variables. Of course, the method to create the specification class instance could have any name which does not conflict with other method names of the implementation. At lines 606-608, the abstraction() method constructs and initializes a new instance of IDictionary$Contract using a map-comprehension expression. The map-

comprehension expression is defined in terms of the implementation array variables, 'keys' and 'values'. The map-comprehension expression defines a mapping between corresponding positions of the arrays. The map comprehension expression defines a map object and is used by the IDictionary$Contract constructor to initialize the map specification variable. In this manner, a correspondence is established between the implementation and specification variables. By providing a method such as abstraction() in the implementation, the specification may be made independent of any particular implementation.

In addition to initialization, it may be advantageous to create and initialize an instance of the specification class immediately prior to invoking any methods of the specification class. This "just-in-time" instantiation enables the state of the implementation to remain synchronized with the state of the specification, regardless of intervening (unspecified) methods or other processing that may alter the state of the implementation in unpredictable ways.

Figures 3-6 involve declarative specifications. As previously described, another type of specification is operational. An operational specification defines a behavior in terms of actions that should be taken. Figures 7-10 illustrate an operational specification and its use in conformance testing.

With reference to the code listing embodiment 700 of Figure 7, an AsmL operational specification of the IDictionary interface begins with a class declaration 702. The specification 700 declares a map object (line 704) and a set of enumeration objects (line 706). An enumeration object comprises data and methods to enable the enumeration of a set, item by item. A client of the IDictionary interface may receive an enumeration object and

employ the object to enumerate the contents of the map object. Line 708 defines an overall constraint condition. This constraint is outside of any method specification and hence is to be applied to all methods of the interface. The constraint specifies that the domain of the map and the set of enumerators cannot comprise any nulls.

Lines 712-724 specify operations that implementations of the set method should perform. Conformance checking is performed by executing the operations of the specification along with the operations of the implementation, and checking the two results with one another. An error occurs if the results don't match.

Lines 714-718 define a first step in the operation of the specification of the set method. A step is a block of operations that occur in parallel. No updates to the state of the specification occur until all operations of a step are complete. In the first step at 716, if the provided key is null, an exception is thrown. Otherwise, at 718 the provided key is associated with the provided value in the map.

Lines 720-722 define a next step in the operation of the specification of the set method. All enumerators are invalidated (e.g. marked as unreliable or useless). The enumerators are invalidated because each enumerator comprises a current location in the map that determines which positions of the map have already been enumerated, and which positions are yet to be enumerated. When the map is altered in mid-enumeration, the enumerator loses its context and must be invalidated.

Line 724 defines a next step in the operation of the set method. The provided value that was associated with the key in the map is returned.

In one embodiment, a new class may be generated embodying operations of the specification. An object instance of this new class may be invoked at strategic points during the execution of the CT-enabled implementation. In this manner, the operations of the implementation and the specification may be executed together, and the results compared, so that the implementation may be checked for conformance with the operational behavior of the specification.

With reference to the code listing embodiment 800 of Figure 8, a class generated to carry out the operations of the specification begins with a class declaration 802. At lines 804-806, an AsmL 'map' object and an AsmL 'set' object are declared in the generated class. The map object associates values or objects with corresponding values or objects in a one-to-one fashion. The set object defines a collection of objects. Although the class is illustrated in C# and pseudocode for simplicity, in one embodiment the class is generated in an intermediate language.

At 808 a set$Pre() method is defined to assert any require conditions (none were present in the specification). At lines 810-818 a method set$Post() is defined to include the operations from the specification of the set method. Any return statements are replaced with an assertion that the result of executing the specification code matches the result provided from the implementation (i.e., ASSERT(result == e)). If the execution of the specification code results in an exception, the exception is checked against the result provided by the implementation at line 818. The type of both exceptions should be compatible.

With reference to the code listing embodiment 900 of Figure 9, an implementation class with conformance checking begins with a class declaration 902. At 904, the

implementation instantiates the IDictionary$Contract class which implements the operational specification 700. At 906 the implementation variables for the hash table arrays are declared. The set() method of the implementation is declared at 908, along with a return variable at 910. At 912 the Invariant() method of the specification object is invoked to enforce any constraints. The set$Pre() method of the specification object is called at 914 to enforce any preconditions. The body of the implementation of the set method is executed inside a try-catch construct at 916-920. Line 920 indicates that any return statements in the implementation body have been replaced with an assignment of the method result to a variable and a jump to line 924. If there was an exception, the catch clause traps it and assigns the exception to the result variable. At 926 set$Post() is called with the result of executing the implementation. The specification code is executed in set$Post() and the results of executing the implementation and the specification are compared. An exception is asserted if the two results don't match. Otherwise, set$Post() returns, and at 928 the constraint condition is enforced again. Finally at 930, the result of executing the implementation is performed; either the result is returned, or if execution resulted in an exception, the exception is thrown.

To reach a common result, the implementation and the specification may need to begin executing from a common starting state. The implementation may provide an initialization method to construct the specification object with the proper starting state. With reference to the code listing embodiment 1000 of Figure 10, the implementation class 1002-1010 comprises a specification object at 1004. Figure 10 contains additional definitions for the class defined in Figure 9. At 1006 an initialization method is defined to set the

specification object with the proper starting state. A constructor 1008-1010 of the implementation class calls the initialization method to properly initialize the specification object.

For the same reasons set forth in the discussion of declarative specifications, it may be advantageous to instantiate the operational specification class (or at least, to resynchronize the state of the implementation and the specification object) immediately prior to invoking any methods of the specification object.

Another challenge in conformance testing arises when there is non-determinism in the specification. As previously described, non-deterministic specifications define ranges or choices within which various acceptable behaviors may take place. Non-deterministic specifications are desirable because they do not specify the implementation behavior down to the finest detail. Non-deterministic specifications define choices for behavior, providing the software implementer with design freedom within those choices.

With reference to the code listing embodiment 1100 of Figure 11, an alternative specification for the set() method 1102-1118 includes a nondeterministic choice 1112. (Compare this specification with the one in Figure 7.) If the provided key exists already in the map, the specification allows the implementation to choose to return either the provided value, or the existing value in the map corresponding to the provided key. If the choice fails for some reason, lines 1115-1116 provide a runtime exception. The choice will always succeeds in the specification; it can however fail in the CT enabled implementation.

Where, as here, the value being non-deterministically chosen is also the return value of the method, a general code pattern can be provided to enable conformance checking of an implementation with the nondeterministic specification. The pattern:

```
choose x in S where p(x)
        R(x)
ifnone
        Q
```

leads to the following conformance-checking code

```
let x = result
if x in S and p(x)
        R(x)
else
        Q
```

Here, p(x) is a condition which the specification defines and which the nondeterministic choice must meet. In other words, "choose x in S where p(x)" means, "make a choice x from the set S, where x satisfies the condition p(x)". The symbols R(x) represent at least one operations to perform if a conforming choice can be found in the set S. The symbol Q represents at least one operation to perform if a conforming choice can not be found in the set S. Either of R(x) and Q(x) could be null (no operations to perform if the choice succeeds or fails, respectively) or "no ops", meaning that the operations are merely

placeholders with no real effect on execution. With reference again to Figure 11, the following code tests for conformance with lines 1112-1116 (the nondeterministic choice) of the specification:

```
let r = result
if r in {map(key), value}
        return r
else
        throw new RuntimeException()
```

The return result of the implementation method is placed into a variable r, as in Figure 4. A test is made to determine if the return result is one of the new value set in the map, or the old map value. If so, the result is returned from the method. Otherwise, the exception is thrown. Thus, the implementation method is tested for conformance with the non-deterministic specification. Note that in this embodiment the implementation is executed first, so that the return result of the implementation is available to test against the specification.

In another embodiment, instead of placing the return result of the method into the variable r, the implementation provides a method to return any non-deterministically chosen value into the variable r.

Another challenge is the conformance checking of execution steps and calls to methods outside of the class to test. Consider a specification in which a method must perform processing and make calls to other methods in a certain order. The called methods,

in turn, may call back into methods of the class that provides the calling method. This situation can lead to re-entrance of the class in manners that change the state of the calling method in unpredictable ways.

To handle this situation, the specification code can be divided into steps. Each section corresponds to a set of operations which can occur concurrently. The steps themselves are ordered, i.e., are specified to occur in a certain sequence. For example, each of the three steps of the specification code listing 700 of Figure 7 may comprise a separate section of a conformance-test enabled object. Herein, the term 'section' refers to one or more instructions which can be executed independently of the instructions of other sections. Sections may be delimited by conditional statements such as if-then statements, case statements, go-to statements, or other control-flow control techniques. Sections may also be encapsulated into methods, functions, etc. A sequencing variable may be employed to execute the sections in an order corresponding to the order of the corresponding steps in the specification.

A new class embodying the sections may check that the operations of the implementation are carried out in the proper sequence, that the implementation makes mandatory calls as specified, and that the state of the implementation state remains in conformance with the specification.

If calls to methods in other classes (henceforth, 'mandatory calls') are indicated in the specification, a check is made to confirm that those calls are made by the implementation. A check is made during each mandatory call to confirm that the state of the implementation remains in conformance with the specification. In one embodiment, a new

class is generated to handle these tasks. (In the examples herein, this new class is a subclass of IDictionary$Contract from Figure 8.) There is one subclass defined per method for which there is a specification. Thus, in the examples herein a single subclass called IDictionary$Contract$Set is generated. All subclasses share a common runtime stack, embodied by the object Stack in the code listing embodiment 1200 of Figure 12. The code 1200 shows a CT enabled implementation 1202-1228 of the specification of Figure 7. An instance of the IDictionary$Contract$Set subclass is generated at 1212. The subclass includes a method, Step() (illustrated fully in Figures 14-16), which performs the following tasks:

- Executes steps of the specification in sections, in the proper order.
- Checks that the implementation made the mandatory calls that were indicated in the specification.
- During each mandatory call, checks that the implementation state remains in conformance with the specification.

In situations where the method under conformance test (in this example, the set method) may be reentered, a stack of instances of the subclass may be maintained by the CT enabled implementation (in this example, IDictionary$Contract). Instead of storing a single instance of IDictionary$Contract$Set in an instance variable, the set$Pre method of IDictionary$Contract may create a new instance of IDictionary$Contract$Set at 1212 and push it onto a stack of such instances (called, for example, setInstanceStack). All references to the subclass in Figure 12 would then be made into references to the instance of the subclass on the top of the stack. Thus, for example, the call to "setInstance.Step()" at lines

1216 and 1228 would become "setInstanceStack.top.Step()". The set$Post method would pop the instance of the subclass from the top of the stack before returning.

In one embodiment, a call to the Step() method is included in the CT enabled implementation class as part of the methods set$Pre() and set$Post.

Figure 13 shows a code list embodiment 1232-1248 of a subclass to conformance test the set method of the IDictionary$Contract class of Figure 12. The subclass contains a member variable, pc (for program counter), which controls the sequence of execution of the sections of Step() corresponding to the specified steps of the set method. The class also contains a member variable corresponding to each parameter of set. These member variables retain their values across separate invocations of Step(). These member variables (key, value) are set by the constructor 1246-1248 of the IDictionary$Contract subclass, and the program counter is initialized to zero.

With reference to the code listing embodiment 1400 of Figure 14, the Step() method is declared 1250 and includes an outer execution loop 1252. The first time Step() is called (by set$Pre()), pc is zero and thus the switch 1254 chooses case 0 at line 1256. A new "stack frame" is created and pushed onto the stack. The program counter is incremented, and the outer loop 1252 executes a switch 1254 to case 1 at line 1258. Instructions 1260-1264 correspond to the first step of the specification 700. A test at 1266 determines if the first step pushed mandatory calls on the stack (the mandatory call itself is not pushed – rather, an identification of the call is pushed, as described further in conjunction with Figure 15). If the first step made mandatory calls, the program counter is incremented and Step() returns. Otherwise, the program counter is incremented by three to execute the instructions

corresponding to the second step of the specification. Any exceptions in the first step cause the sequence variable pc to be set to the value END (indicating an end to execution of the specification sections) before Step() returns.

In this example, the first step of the specification does not specify any mandatory calls. Thus, lines 1276-1280 (where the program counter has the values 2 or 3) are not executed. Execution skips to the second section of the code (case 4), corresponding to instructions of the second step of the specification.

With reference to the code listing embodiment 1500 of Figure 15, the second section of the code (lines 1282-1288) stores at 1288 an identification of each mandatory call that the specification indicates the implementation should make. Lines 1286 indicate that a mandatory call should be made to the Invalidate() method of each outstanding enumeration object (refer back to the description of Figure 7 for the purpose and function of enumerator objects). Each specified mandatory call results in the addition of a "call signature" to the current stack frame. It does not push any new elements on the stack, but instead adds information to the frame that is on the top of the stack. A call signature includes 1) the identity of the object which comprises the Step() method, 2) the identity of the object on which the mandatory call is invoked, 3) the name of the mandatory call, 4) the values of any parameters to the mandatory call (in this case, null is passed because the Invalidate() method has no parameters). Other embodiments might comprise other information to uniquely identify each call. Because the stack frame now includes the signatures of mandatory calls, pc is incremented by one (1) and control is returned to the conformance tested implementation (see lines 1266-1274). To conform to the specification, the CT enabled

implementation must now execute the mandatory calls identified in the stack frame. Referring to the code listing embodiment 1700 of Figure 17, the software comprising the mandatory call (lines 1350-1364) is adapted to:

- Locate the signature of particular invocation of the mandatory call in the stack frame (line 1354).
- Call Step() a first time on the sub-class instance corresponding to the method that invoked the mandatory call (line 1356).
- Call Step() a second time on the sub-class instance corresponding to the method that invoked the mandatory call (line 1362).
- Remove the call signature from the stack frame (line 1362).

Referring back to Figure 15, the first call to Step() at 1354 results in the execution of lines 1292-1294 (case 5). A check is made to determine that the state of the CT enabled implementation conforms to the specification at the time of the mandatory call. In addition to testing the constraint condition, the "strongest post-condition" of the prior step of the specification is also tested. The strongest post-condition may be determined in manners well known in the art, and in this case comprises the condition "map(key)==value". The variable pc is then incremented so that the next time Step() is called, lines 1296 are executed (case 6). Control is returned to the mandatory call method, which calls Step() a second time at 1362 (Figure 17). During this second call to Step(), a check is made to determine whether the frame at the top of the stack comprises additional mandatory call signatures (e.g. if the specification step under test specified additional mandatory calls). If so, the sequence variable pc is decremented so that lines 1292-1294 (case 5) are executed again to check the

state of the CT enabled implementation the next time the CT enabled implementation makes a mandatory call as part of the same specification step.

To summarize, the CT enabled implementation calls Step() upon being invoked and before returning. For each specification step that comprises a mandatory call, Step() yields (returns control to the CT enabled implementation) to enable the CT enabled implementation to make the conforming mandatory call (or calls). Step() is called twice by the software that implements the mandatory call. The first call confirms that the implementation state conforms to the specification. The second call records the termination of the mandatory call and resets the sequence variable, pc, so that Step() tests for additional specified mandatory calls, if any. Thus, in one embodiment the implementation of the mandatory call method plays an active part in the conformance testing of the implementation that calls it.

With reference to the code listing embodiment 1600 of Figure 16, lines 1298-1316 perform substantially the same purpose for the third step of the specification as lines 1282-1296 performed for the second step of the specification (e.g. confirming mandatory calls by the CT enabled implementation, checking the state of the CT enabled implementation during the mandatory calls, etc.). Once the last mandatory call has been confirmed, the program counter is set to END at 1316. When Step() is called a final time by set$Post (see Figure 12, line 1228), lines 1318-1330 (case END – see Figure 16) are executed. These lines check that:

- There are no mandatory calls unaccounted for and left on the stack (line 1320)

- The final result of executing the specification is the same as the result of executing the CT enabled implementation (lines 1324-1326). Note that in this embodiment it is the responsibility of set$Post in the CT enabled implementation to set the result in a manner that it can be accessed from Step() (see line 1222 of Figure 12).
- Remove the stack frame for the method under test (line 1328).

With reference to Figure 18, an apparatus embodiment 1400 for practicing embodiments of the present invention comprises a processing unit 1402 (e.g., a processor, microprocessor, micro-controller, etc.) and machine-readable media 1404. Depending on the configuration and application (mobile, desktop, server, etc.), the memory 1404 may be volatile (such as RAM), non-volatile (such as ROM, flash memory, etc.) or some combination of the two. By way of example, and not limitation, the machine readable media 1404 may comprise volatile and/or nonvolatile media, removable and/or non-removable media, including: RAM, ROM, EEPROM, flash memory or other memory technology, CD-ROM, digital versatile disks (DVD) or other optical storage, magnetic cassettes, magnetic tape, magnetic disk storage or other magnetic storage devices, or any other medium which can be used to store the desired information to be accessed by the apparatus 1400. The machine readable media 1404 may be implemented in any method or technology for storage of information such as computer readable instructions, data structures, program modules or other data. Such instructions and data may, when executed by the processor 1402, carry out embodiments of methods in accordance with the present invention.

The apparatus 1400 may comprise additional storage (removable 1406 and/or non-removable 1407) such as magnetic or optical disks or tape. The apparatus 1400 may further comprise input devices 1410 such as a keyboard, pointing device, microphone, etc., and/or output devices 1412 such as display, speaker, and printer. The apparatus 1400 may also typically include network connections 1420 (such as a network adapter) for coupling to other devices, computers, networks, servers, etc. using either wired or wireless signaling media.

The components of the device may be embodied in a distributed computing system. For example, a terminal device may incorporate input and output devices to present only the user interface, whereas processing component of the system are resident elsewhere. Likewise, processing functionality may be distributed across a plurality of processors.

The apparatus may generate and receive machine readable instructions, data structures, program modules or other data in a modulated data signal such as a carrier wave or other transport mechanism. The term "modulated data signal" means a signal that has one or more of its characteristics set or changed in such a manner as to encode information in the signal. This can include both digital, analog, and optical signals. By way of example, and not limitation, communication media includes wired media such as a wired network or direct-wired connection, and wireless media such as acoustic, RF, infrared and other wireless media. Communications media, including combinations of any of the above, should be understood as within the scope of machine readable media.

In view of the many possible embodiments to which the principles of the present invention may be applied, it should be recognized that the detailed embodiments are

illustrative only and should not be taken as limiting in scope. Rather, the present invention encompasses all such embodiments as may come within the scope and spirit of the following claims and equivalents thereto.